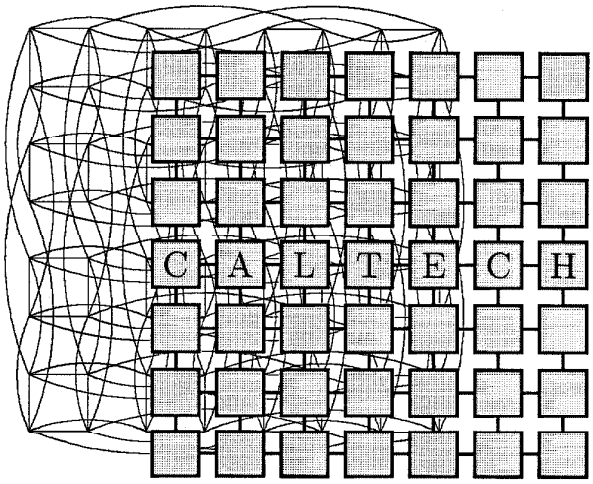

Submicron Systems Architecture Project
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

The C Programmer's Abbreviated Guide to Multicomputer Programming

Charles L. Seitz, Jakov Seizovic, Wen-King Su

Caltech Computer Science Technical Report
Caltech-CS-TR-88-1
19 January 1988
(Revision 1 17 April 1989)

The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745, and in part by grants from Intel Scientific Computers and Symult Systems, Inc.

The C Programmer's Abbreviated Guide to Multicomputer Programming

Charles L. Seitz, Jakov Seizovic, Wen-King Su

Department of Computer Science
California Institute of Technology

Caltech Computer Science Technical Report
Caltech-CS-TR-88-1

19 January 1988
(Revision 1 — 17 April 1989)

Abstract: This *abbreviated* programmer's guide describes the essentials of writing multiple-process message-passing C programs on UNIX hosts under a runtime system called the *Cosmic Environment* (version 7.2), and on multicomputers under a node operating system called the *Reactive Kernel*. It is our intention that a person who is already familiar with C and UNIX, and with the formulation of concurrent computations for multicomputers, will find a single reading of this guide to be sufficient preparation to start writing multicomputer programs. The Appendices include reference material and digressions.

Please send suggestions about this document to `chuck@vlsi.caltech.edu`, and bug reports to `cube@vlsi.caltech.edu`.

The research described in this report was sponsored in part by the Defense Advanced Research Projects Agency, DARPA Order number 6202, and monitored by the Office of Naval Research under contract number N00014-87-K-0745; and in part by grants from Intel Scientific Computers and Symult Systems, Inc.

Contents

The C Programmer's Abbreviated Guide to Multicomputer Programming

1. Introduction

When abstracted to the level of the application programmer, a *multicomputer*, also referred to as a *message-passing concurrent computer*, consists of N computers, called *nodes*, connected by a message-passing communication network:

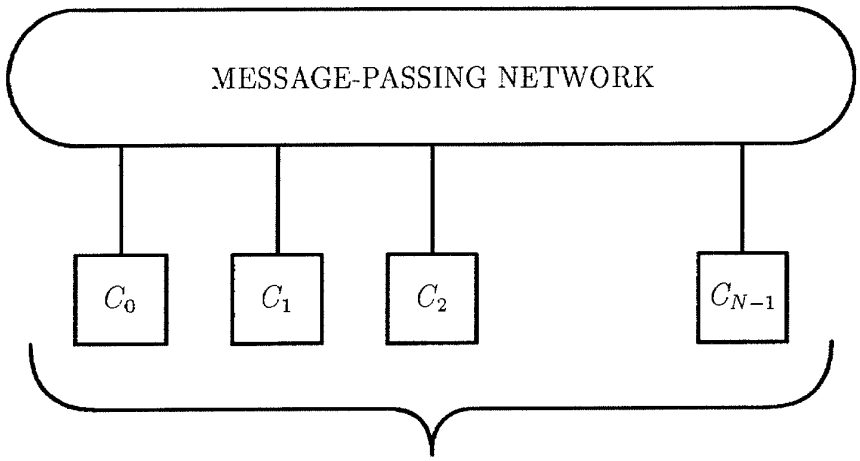

Figure 1: Programmer's Model of a Multicomputer

Each of the N node computers executes programs concurrently, and coordinates its activities with the other nodes only by sending and receiving messages. Multicomputers are *distributed-memory* systems. In contrast with the other principal type of MIMD architecture, the shared-memory multiprocessor, a multicomputer is organized with the processor(s) in each node computer having a memory that is physically separate and logically private from those of the other node computers. There are no global memory addresses; rather, each node has its own private memory address space. The fixed numbering of the nodes, $0, 1, \ldots, N\text{-}1$, together with the unique numbering of the processes within each node, establishes globally unique identifiers for processes; hence, a global name space.

The occasional appearance in this guide of the word "cube" to refer to a multicomputer is a carryover from first-generation multicomputers — the Cosmic Cube and its commercial descendants — that use software-controlled routing on a binary n-cube (hypercube) network to connect $N = 2^n$ nodes. Second-generation multicomputers employ message-routing hardware that makes the topology of the message-passing network practically invisible to the programmer.

A multicomputer also includes one or more nodes with interfaces to a local area network, and/or one or more computers with interfaces to both the multicomputer's message-passing network and a local-area network. Either of these hardware configurations allows messages

to be passed between multicomputer nodes and network *hosts*, such as the user's workstation computer.

The Cosmic Environment (CE) and Reactive Kernel (RK) were developed to support a message-passing programming environment on network hosts and multicomputer nodes, respectively. However, the CE/RK programming environment is quite portable, and allows the same message-passing programs to be run on other architectures.

The CE system consists of a set of dæmon processes, utility programs, and libraries. It can be used as a stand-alone system for running message-passing programs on collections of network-connected UNIX hosts, and can also handle the allocation of, and interfaces to, one or more multicomputers. When interfaced to a multicomputer, it supports uniform communication between host and node processes.

RK is a new node operating system for multicomputers. Although its design and implementation were motivated principally by the development of second-generation multicomputers, it was first developed on the original Cosmic Cubes. RK was later ported to the Symult Series 2010, a second-generation multicomputer that was developed as a joint project between Symult Systems, Inc. (formerly the Ametek Computer Research Division) and our research group in the Caltech Computer Science Department. The same programming environment is also supported on the Intel iPSC/1 and iPSC/2 multicomputers with the NX node operating system through the use of a compatibility library. A port of RK to run as the native node operating system on the Intel iPSC/2 is currently in progress.

The only difference between running message-passing programs on UNIX hosts under the CE system and on multicomputers under RK is their process-creation functions. In order to be able to simulate multicomputers precisely, including the process-creation functions, the CE system includes a feature, called *ghost cubes,* that allows a collection of network-connected UNIX hosts to masquerade as a multicomputer. Thus, multicomputer programs written in this environment run interchangeably on the Cosmic Cubes, Intel iPSC/1, Intel iPSC/2, Symult Series 2010, and ghost cubes.

In addition to its pedagogical intentions, this guide serves as a reference for the base definition of the CE/RK system; that is, for those functions that must be supported in any implementation. Most implementations will include additional features and functions, such as a set of UNIX-compatible functions, which we refer to as the *UNIX Annex* of a given implementation. Of course, we impose no restriction on how any of these functions are implemented; what may be implemented under one operating system as a native function may be implemented under another as a library function. Where it is necessary or useful to mention the extensions and idiosyncrasies of particular implementations, these and various other digressions are contained in the Appendices.

2. The Process Model

The basic unit of computation in this environment is a *process.* A process is an *instance* of a program — for our purposes, a program written in C — that includes statements that cause messages to be sent and received. This notion of a process is essentially similar to processes in multiprogramming operating systems such as UNIX, and is also a useful abstraction of the architecture of a multicomputer. Each process has its own private address space, just as each node has its own memory. Processes, like nodes, communicate by message passing.

A copy of the node operating system, RK, runs on each node. It supports multiprogramming within a single node, such that each node may contain many processes. The number of concurrent processes involved in a single computation can accordingly be much larger than N. Computations are performed by the concurrent execution of processes distributed through the multicomputer nodes, and one or more *host processes.*

Although message passing is about one hundred times slower in local area networks than in today's multicomputer message-passing networks, CE provides exactly the same message-passing and other functions for UNIX processes that RK provides for node processes. Thus, CE and RK together provide uniform communication between processes independent of the multicomputer node or network host on which they happen to be located. It is a principle and a formal property of the entire system — within the limits of the computation being deterministic and not exceeding available storage sizes — that the results of a computation do not depend on the way in which the processes are distributed.

Computations formulated for this multiple-process message-passing environment can be run with or without a multicomputer. The processes could all be run on a single host, distributed across a number of network-connected hosts, distributed amongst the processors of a shared-memory multiprocessor, distributed across the nodes of a multicomputer, or combinations of these possibilities.

The entire set of processes involved in a single computation is called a *process group.* A process group and a (possibly empty) set of multicomputer nodes are allocated by the user with the host utility `getcube`. Since the first-generation multicomputers are all binary n-cube connected systems that are shared by recursive partitioning into sub-cubes of lower dimension, the convention is to specify the "cube" size desired by its dimension, $n = \log_2 N$. The arguments to `getcube` specify options of the process group and the size and model of multicomputer to be allocated. For example, one can run:

```
% setenv cube "group first"
% getcube 2 non
2D non-cube allocated
```

to allocate a process group of host processes only, for which the dimension of the cube is not relevant, or:

```
% setenv cube "group second"
% getcube 3 ghost
3D sub-cube allocated
```

to allocate a process group together with a `ghost` cube, whose 2^3 nodes are distributed across

up to eight computers on a local area network, or:

```
% setenv cube "group third"
% getcube 6 cosmic
6D sub-cube allocated
```

to allocate a process group together with a cosmic cube with 2^6 nodes. The process group has a name in the form {groupname username}. In the three examples above, the groupname is set by the contents of an environment variable, cube; in particular, by the word that follows the keyword group. After creating a process group, the getcube process puts itself into the background and remains there until the freecube program is run.

Running the peek utility reveals the way in which all the multicomputers on a network are allocated, as well as one's own host processes:

```
% peek
CUBE DAEMON version 7.2, up 19 days 10 hours on host sol

{               }  5d ipsc cube   , b:0000 [   mars :iPSC d7]  5.4h
{CONCISE lena   }  5d ipsc cube   , b:0020 [   mars :iPSC d7]  1.0h
{     TV wen-king}  6d ipsc cube   , b:0040 [neptune :iPSC d7]  1.3h
{               }  3d cosmic cube, b:0000 [  venus fly trap]  5.4h
{  ngai3 ngai   }  4d ipsc2 cube  , b:0000 [ triton iPSC2   ]  3.0h
{Zipcode skjellum} 16n s2010      , b:000c [mercury :ginzu  ]  7.6m
{ Mosaic jakov  }  48n s2010      , b:000d [  metis :ginzu  ]  2.2h
{ second chuck  }   8n ghost cube , b:0000 [ icarus :mimic  ] 12.0s
{               }   8n ghost cube , b:0008 [ saturn :mimic  ]  1.2d
{  third chuck  }   6d cosmic cube, b:0000 [ icarus 6-cube  ]  6.0s
{  first chuck  }   2d non cube   , b:0000 [ icarus         ] 20.0s

GROUP {third chuck}  TYPE reactive  IDLE 0.0s

( -1  -1)    SERVER    0s    0r   0q   [icarus 5051]  3.0s
( -1  -2)   FILE MGR   0s    0r   0q   [icarus 5052]  3.0s
(--- ---)   CUBEIFC    0s    0r   0q   [ ceres 8327]  5.0s
```

Multicomputers are normally not time shared, but *space shared.* For example, the 128-node iPSC d7 is allocated, as shown in the first three lines of the peek display, with one user (wen-king) running a computation on a 6d ipsc cube (64 nodes), another user (lena) on a 5d ipsc cube (32 nodes), and the remaining 5-cube being free. Of course, the user with 64 nodes cannot distinguish these 64 nodes from a separate 64-node multicomputer. The logical node numbers will be $0, 1, \ldots, 63$, regardless of their physical location in the multicomputer. Multicomputers that do not use binary n-cubes, such as the Symult Series 2010 and ghost cubes, will indicate the number of nodes rather than the dimension of a binary cube. A number of nodes that is not a power of two can be allocated on such systems with a command such as getcube 3n ghost. Space-sharing has proven to be most appropriate for multicomputers, since it allows a user to select a number of nodes that is appropriate to the number of processes

and the load-balancing characteristics of a particular computation, produces predictable run times, and does not allow one user's processes to upset the load balance of another process group.

The peek display also shows the host processes in the process group, and the number of messages sent, received, and queued by each host process. The SERVER process is started by getcube, and is available to access programs stored in the host file system and to print messages on the user's screen. The FILE MGR services standard I/O functions in node processes. The CUBEIFC process relays messages between the network and any allocated multicomputer.

A number of other host utilities are described in Appendix C.

3. Process IDs, Placement, and Spawning

Every process within a process group has a unique *ID*, which is represented as an ordered pair: (node, pid). For node processes, node is a C integer in the range $0 \ldots N\text{-}1$. The pid of a user process within a node is a C integer in the range $0 \leq$ pid $\leq$ MAXUPID. For host processes, node is generally equal to the constant HOST, and a unique ID is maintained by the host runtime system. For example, the peek display illustrated above shows that the IDs of the SERVER and FILE MGR processes are (-1, -1) and (-1, -2), respectively; thus, HOST is -1. System processes and user processes that enter the environment without specifying a pid are assigned negative pids, while user processes ordinarily have non-negative pids.

The philosophy behind the system's process-spawning mechanisms is to give the programmer explicit control of process placement through the arguments of dynamic process-creation functions. Process placement may thus be decided either while the program is being written or dynamically during execution. The same set of dynamic process-creation functions serve to execute the placement choices in either case. Processes in execution may freely spawn and kill other processes.

Once a process is spawned, it will not spontaneously migrate to another node. Thus, it is most efficient simply to retain the physical location of a process as a part of its unique ID. An implicit premise of multicomputer architectures is that costly communication mechanisms for dynamically binding computing resources to runnable objects, as used in multiprocessors or in the dataflow model, are rarely necessary. It is instead sufficient for most problems to establish a binding of a process to a node that will persist for the life of the process. Of course, this does not rule out changing the bindings. Although the CE/RK environment does not directly support the "automatic" relocation of processes, environments that support dynamic relocation of processes are fairly easily layered on top of the CE/RK environment by employing a distributed data structure that maps references to IDs, and by using the dynamic process-spawning features.

In either a host or node process, one may spawn a *node* process from a compiled program with the function:

```
spawn("filename", node, pid, "mode");
```

where "filename" is a character string argument that specifies a file relative to the working directory of the getcube program that allocated the process group. The node and pid arguments specify the ID of the process to be created. The "mode" argument should be an

empty string. The spawn function returns 0 if it is successful, or a non-zero error code if it is unsuccessful. Process spawning will fail if the specified file does not exist, if the node does not exist or has insufficient memory, or if the pid is already in use or outside of the allowed range. Thus, one might use spawn in the following form if one were to allow for the possibility of it failing:

```
if (errcode = spawn("sinker",5,3,"")) {
    print("failed spawn, code %d",errcode);
    exit()
}
```

The spawn function returns only after the process exists or the operation has failed. When a process is successfully spawned, a startup routine calls main(), just as in other C environments.

One can get rid of a node process with the function:

```
ckill(node, pid, "mode");
```

where the "mode" parameter should be an empty string.

If the node parameter of the spawn function is specified as -1, a process will be created in every node. Similarly, if the node parameter of ckill is -1, all node processes with the specified pid will be killed. The ckill function will also take a pid argument of -1 to apply to all processes in a given node. Executing ckill(-1,-1,"") will kill all processes in all nodes.

It is perfectly all right to use spawn and ckill in host processes, although their effect is only on node processes. Executing the spawn function in host processes is the way in which processes are initially loaded into nodes.

4. Messages

A *message* is the logical unit of information exchange between processes. A message may be any number of bytes in length, from 0 to any size that will fit in the memory of the nodes of the sending and receiving processes. Although messages may be routed by different protocols according to their length or destination, these differences are entirely invisible to the application programmer.

What is necessary for one process to send a message to another process is that the sending process have *reference* to the receiving process. References are represented in message functions as IDs. By the term *process structure* we mean the set of processes within the process group, together with each process's references to other processes. The process structure is naturally depicted as a directed graph with vertices representing processes and with arcs representing references. The arcs can also be visualized as virtual communication channels, with messages traveling along the arcs, such as the example shown in Figure 2.

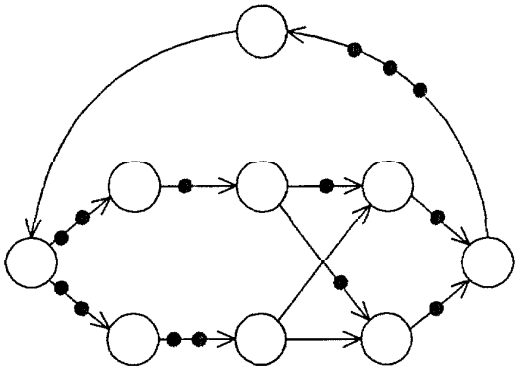

Figure 2: An example of a Process Structure Graph

The message system has a number of properties that are important to the programmer. The main property is that messages are queued as necessary in the sending node, in transit, and in the receiving node; and accordingly have arbitrary delay from sender to receiver. However, between *pairs* of communicating processes, message order is preserved; *ie,* if process A sends two messages in sequence to process B, they will arrive at an arbitrary time later and in the same order as they were sent. Thus, in Figure 2, the messages might be thought of as "traveling" at some arbitrary speed, but one message is not able to pass another on a reference arc.

Every message carries some imbedded self-representing information, including reference to its destination process and the message length. Some implementations may include additional information in the message, such as reference to the sending process. Such information is often useful when debugging programs, because it allows one to determine the origin of a rogue message.

Messages are sent from and received into dynamically allocated memory that can be accessed both by the user process and by the message system. Message buffers are arrays of characters with no presumed structure, and the functions that return pointers to message

buffers return maximally aligned pointers of type `char*`.

Message space can be allocated by:

```
p = xmalloc(length);
```

where the `length` is specified in bytes, and can be deallocated by:

```
xfree(p);
```

These functions are identical in usage to the UNIX `malloc` and `free` functions, but in most implementations manage a different area of memory.

When a message has been built in a block pointed to by `p`, its contents can be sent as a message to process `(node, pid)` by:

```
xsend(p, node, pid);
```

The `xsend` function also deallocates the message space; that is, `xsend` is like `xfree`, except that it also sends a message. Thus, there is no need for blocking or for feedback that the message has been sent. When the function returns, the message block is gone.

A messages is received by:

```
p = xrecvb();
```

As indicated by the `b` at the end of `xrecvb`, this is a *blocking* function that does not return until a message has arrived for the process. The execution of the `xrecvb` function is just like allocating a message buffer with `xmalloc`, except that the length and initial contents of the block allocated is determined by the length and contents of the message received. Once the message is no longer needed, the allocated space should be freed. Of course, the message space can be freed with `xfree`, but it can also be freed by `xsend` if there is a message of the same length to send. It frequently happens in message-passing programs that a message that is received is simply modified by a computation and then sent on to another process.

If the length of a message that is received is not known, it can be determined by:

```
length = xlength(p);
```

where `length` is in bytes.

The non-blocking receive function is called `xrecv`. It is required only for advanced applications in which a process may need to *probe* for another message without giving up the right to continue execution. The usage of the `xrecv` function is identical to `xrecvb`; however, it may return a NULL pointer if there is no message queued for this process. The `xrecv` function should not be used casually; `xrecvb` is the "normal" receive function. However, `xrecv` is the fundamental receive primitive; `xrecvb` could be expressed in terms of `xrecv` by a busy-wait loop:

```
char *xrecvb() { char *p; while(!(p = xrecv())); return(p); }
```

while `xrecv` cannot be expressed in terms of `xrecvb`.

The behavior of the `xrecv` function allows one to write programs that can do other work while waiting for a message; for example:

```
while (1) {
    if (p = xrecv()) digest(p);
    else do_other_work();
}
```

In such usage, the `digest(p)` and `do_other_work()` functions should return after a bounded time to call `xrecv` again, because calling `xrecv` or `xrecvb` when the next message in the node's receive queue is for another process allows the kernel to save the state of the process that called for the message and start running the process that has a message waiting for it. The appearance of `xrecv` or `xrecvb` in the code marks *choice points* for switching the execution to another process, and it is in this sense that the scheduling is *reactive* or *message driven*. So long as a process is making progress, the kernel does not necessarily force a context switch unless some exceptional system event occurs. Thus, it is possible for a process that is running for a long period without executing a `xrecvb` or `xrecv` to block all other user processes on the same node from execution.

Since `xsend(p, node, pid)` frees the block pointed to by `p`, sending the same message to several destinations with `xsend` would involve copying. Thus, a *multiple-destination send function* has been defined:

```
xmsend(p, cnt, destinations);
```

where `p` points to the message to be sent, `cnt` is an integer that specifies the number of messages to be sent, and `destinations` is an array of at least 2*`cnt` integers in the form `{node1, pid1, node2, pid2, ...}`. Just as with `xsend`, the `xmsend` function is equivalent to `xfree`, except for the side-effect of sending the messages.

The `xmsend` function is implemented to exploit whatever efficient hardware mechanisms may exist in a given system to accomplish multiple-destination sends; but, lacking any such mechanisms, it is implemented as a library function that performs the necessary copying and multiple calls to `xsend`.

5. Other Functions

The CE/RK environment has a number of other functions that are not directly related to spawning processes or to sending and receiving messages:

`mynode()` returns the number of the logical node in which the process is running.

`mypid()` returns the process's own `pid`.

`nnodes()` returns the number of nodes, N.

`cubedim()` returns the "dimension" of the logical "cube" in which the program is running — *eg,* $\log_2 N$ if N is a power of two, or $\lfloor \log_2 N \rfloor$ if N is not a power of two.

`clock()` returns a sampling of a real-time value, which is not necessarily the same for all nodes. The value returned is an `unsigned long` that increments at approximately 1ms intervals. Use of the `clock` function to synchronize different parts of a computation is strongly discouraged; it is intended only for making performance measurements.

`led(x,n)` turns the light-emitting diode `n` for that node off or on for `x` false or true, respectively. Different multicomputers may have different numbers of light-emitting diodes or other node displays. More generally, this function allows the user to copy the value of `x` into output device `n`, if device `n` exists; otherwise, the function has no effect.

`print(...)` is similar to UNIX `printf`, as illustrated in the following sections.

`execute(...)` is a variant of `print` that, instead of printing the resulting string on `stdout`, provides it as an argument to a `system` function in the `SERVER` process.

`exit()` terminates the process. An `exit()` need not be included as the last statement in `main()`. The "return" from `main()` includes an `exit()`.

One small complication in the communication between hosts and nodes, and between different types of hosts, is that their byte-order convention is not necessarily the same. Multicomputer interfaces are designed so that the `char` type is always compatible, but, unfortunately, different processors use different conventions for packing and unpacking other types into byte streams. We would like to make programs as portable as possible; accordingly, a set of functions is provided for conversion in the host processes between the particular host's data representations and the multicomputer node's data representations:

type	*host-to-cube*	*cube-to-host*
`short *sp`	`htocs(sp,cnt)`	`ctohs(sp,cnt)`
`long *lp`	`htocl(lp,cnt)`	`ctohl(lp,cnt)`
`float *fp`	`htocf(fp,cnt)`	`ctohf(fp,cnt)`
`double *dp`	`htocd(dp,cnt)`	`ctohd(dp,cnt)`

The `cnt` argument specifies how many of this particular data item to convert. These functions are defined as null macros for node processes, so it does not cost any execution time to include them.

Some multicomputers provide additional functions that are compatible with UNIX functions of the same name. We refer to this welcome but unlegislated extension of the CE/RK functions as the *UNIX Annex*. Of course, host processes and ghost cube node processes always have the full suite of UNIX functions supported on that particular host.

A portable support package for the C-language standard I/O library (`stdio`) is included in CE, and can be used by any multicomputer having a CE interface and the reactive primitives, including the Cosmic Cube, Intel iPSC/1, Intel iPSC/2, and Symult Series 2010. Programs using `stdio` should include the header file `stdio.h`, as usual, and may use the following functions:

`clearerr fclose fdopen fflush fgetc fgets fopen fprintf fputc fputs`

`fread freopen fscanf fseek ftell fwrite getchar gets getw printf`

`putchar puts putw rewind scanf setbuf sprintf sscanf ungetc`

A secondary server process, `FILE MGR`, on the `getcube` host handles file I/O requests; path names used by the node programs are relative to the working directory of the `getcube` process. `stdout` and `stderr` are directed to the output of the `getcube` process; `stdin` is not available. This package may be slow, but it is quite usable for multicomputer systems that do not have their own built-in file systems.

6. Compiling

In most systems, the programs used for compiling and running programs in the CE/RK environment are kept in directory `/usr/cube`, which should be in the user's search path. Host and node programs must include the header file `<cube/cubedef.h>`, and must be compiled with the library `-lcube`.

Host and node programs are compiled with compilers that have the same usage as `cc`:

- `cch` for CE host processes. One may also use `cc` if the necessary header files and libraries are installed in the system directories.
- `ccgh` for ghost cube node processes. The executables are appended with the `.gh` suffix.
- `cccos` for Cosmic Cube node processes. The executables are appended with the *`.cos`* suffix.
- `ccipsc` for iPSC/1 node processes. The executables are appended with the `.ipsc` suffix. The `ccipsc` program runs a remote compilation on the 286/310 iPSC/1 host.
- `ccipsc2` for iPSC/2 node processes. The executables are appended with the `.ipsc2` suffix. The `ccipsc2` program runs a remote compilation on the 386/301 iPSC/2 host.
- `ccs2010` for Symult Series 2010 node processes. The executables are appended with the `.s2010` suffix.

The choice of compiler determines not only what kind of code is generated, but also which version of the `cube` or other library is linked with the program. Of course, `cch` and `ccgh` run `cc` for that particular host, and will produce different object code when run on different machines. The executable programs left in the file system of a particular machine will thus be appropriate for host processes or ghost cube node processes that run on that machine.

7. A Confidence-Building First Program

Since you are already a C programmer, you know more than enough to write a program for a node or host process. Following the excellent example of Kernighan and Ritchie's introductory "Hello, world" program, you might enter your favorite text editor and compose a program such as:

```
#include <cube/cubedef.h>

main()
{
    print("Hello, world, from (%d, %d)", mynode(), mypid());
}
```

The cube's `print` function is much like the UNIX `printf`; `mynode` and `mypid` are functions that return values representing the process's own `node` and `pid`. We can compile this program as a host process:

```
% cch -o hello hello.c -lcube
```

Running it will result in:

```
% hello
{group chuck} does not exist
```

because CE cannot figure out what to do with the first `cube` library function that is called. Instead, we need a process group. We can allocate a process group with `getcube`, in this case specifying a `non` cube with a ridiculous dimension of 12:

```
% getcube non 12
12D non-cube allocated
% hello
-1,-7: Hello, world, from (-1, -7)
% freecube
Cube space deallocated
```

Now we are getting somewhere. You can see that the `print` function is slightly different from UNIX's `printf`, in that it starts the line with the ID of the process executing the `print` and adds its own carriage-return (`\n`). If we wanted this process to have a specific ID, instead of the `(-1,-7)` ID that was arbitrarily assigned, we could specify it by adding a call to a `cosmic_init(node,pid)` function in the program. The use of the `cosmic_init` function is illustrated in later examples.

How do we run a program that has been compiled as a node process on a real multicomputer? This task does not involve much more than it did to run the program on the host. It is just a sequence of (1) allocating a process group with a multicomputer, (2) spawning the program as a process, and then, when you are done with it, (3) freeing the process group and multicomputer.

Although we could load our `hello` program by writing a host program that executes the `spawn` function, there is already a utility program of the same name that will do this task for

us. The spawn program has the usage:

spawn *filename [node [pid [mode]]]*

The *node* and *pid* numbers are specified in decimal. The *mode* is usually left blank. So, having remembered to compile hello.c for a Cosmic Cube:

```
% cccos -o hello hello.c -lcube
```

we proceed:

```
% getcube cosmic 3
3D sub-cube allocated
% spawn hello 7 3
hello spawned successfully in node 7, pid 3
7,3: Hello, world, from (7, 3)
```

Just as with the spawn function, if the *node* parameter given to the spawn program is -1, the process is spawned in all nodes. To get a feeling of power, we might try the broadcast-spawning option:

```
% spawn hello -1 55
hello loaded in all nodes, pid 55
2,55: Hello, world, from (2, 55)
1,55: Hello, world, from (1, 55)
3,55: Hello, world, from (3, 55)
6,55: Hello, world, from (6, 55)
5,55: Hello, world, from (5, 55)
7,55: Hello, world, from (7, 55)
4,55: Hello, world, from (4, 55)
0,55: Hello, world, from (0, 55)
% freecube
Cube space deallocated
```

Grateful now that we didn't get a 6-cube, we run the freecube utility in order to release the cube for another user.

8. A Host Process Example

The user of a multicomputer program does not want or need to know all of the manual steps of getting a multicomputer, spawning processes, and freeing a multicomputer. These operations are usually performed by host processes, so that a multicomputer program can be made to look to the user just like any other program. As a brief example, we might make a simple node program, `pong.c`, that does nothing more than to print `"I'm here and gone"` and then send an empty message to process `(HOST,0)` to indicate that it is done:

```
#include <cube/cubedef.h>

main()
{
    print("I'm here and gone");
    xsend(xmalloc(0), HOST, 0);
}
```

The following host program, `ping.c`, acquires a 0-cube (1 node), enters the environment with the ID `(HOST,0)`, spawns the process above, waits to receive the message, and, finally, frees the cube and exits.

```
#include <cube/cubedef.h>

main()
{
    if (system("getcube ipsc 0"))     /* create process group  */
        exit();
    cosmic_init(HOST,0);              /* enter the environment */
    spawn("pong",0,0,"");             /* spawn node processes  */
    xfree(xrecvb());                  /* catch the message     */
    system("freecube");               /* release the cube      */
}                                     /* and exit              */
```

Running this host program produces:

```
% ping
0D sub-cube allocated
pong spawned as (0,0).
0,0: I'm here and gone
Cube space deallocated
```

9. A Message-Passing Program Example

Our next example is a program for seeing how quickly a multicomputer can pass messages between randomly selected nodes. A host process sends an initial message whose first four bytes contain a `long` that represents the number of random hops the message is to take. Each node process tests whether the count received in a message is zero; if it is non-zero, the process decrements the count and passes the message on to a process in a randomly selected node. When the count reaches zero, the node process sends the message back to the host process, which displays the elapsed time. It is somewhat of a game to see how fast the node processes can be made to react to and process a message.

The "straight-line" host program is explained by the comments:

```
#include <cube/cubedef.h>

main(argc,argv)
    int argc;
    char **argv;
{
    unsigned long hops, len, elapse, otime, *p;

    if(argc != 3) { puts("arg: <hops> <len in bytes>"); exit(-1); }

    cosmic_init(HOST,0);                    /* Enter CE as process (HOST,0) */

    hops  = atoi(argv[1]);                  /* argv 1 = the hop count.      */
    len   = atoi(argv[2]);                  /* argv 2 = length of message.  */
    if(len < sizeof(long))                  /* Need enough bytes to carry   */
          len = sizeof(long);               /* a long (the hop count).      */

    p = (unsigned long *) xmalloc(len); /* Get msg of the right size.   */
    *p = hops;                          /* Put the count value in msg.  */
    htocl(p,1);                         /* Byte order conversion.       */

    otime = clock();                    /* Record starting time in ms.  */
    xsend((char*)p,0,0);                /* Send the message.            */
    xfree(xrecvb());                    /* Get the message and free it. */
    elapse = clock() - otime;           /* Compute elapsed time.        */

    printf("Time for sending a %d byte msg %d hops was %d ms\n",
                              len,       hops,       elapse);
}
```

The only difficulty in writing the node process is in deciding how to generate a pseudorandom number sequence that will be different for each process. One approach is to use the node number returned by the `mynode` function as the seed for a random-number generator. Here

we shall use a random-bit-generating function from the cube library. The code is otherwise explained by the comments:

```
#include <cube/cubedef.h>

extern long ranbits();
int node, blink, bits, seed;
unsigned long *p;

main()
{
    seed  = mynode();           /* Seed for the random bit generator.    */
    bits  = cubedim();          /* Bits to get from random bit generator*/
    blink = 0;                  /* Stores current led state in its LSB. */
    set_ranbits((long)seed+1);  /* Seed the random bit generator.        */
    ranbits(3000);              /* Scramble the random bit generator.    */

    while(1) {
        led(blink ^= 1, 0);             /* Toggle the LED0 as I work.    */
        node = ranbits(bits);           /* Pick a node for next send.    */
        p = (unsigned long *) xrecvb(); /* Wait for the mesg to arrive. */
        if(p[0]--)                      /* Decrement hop count.          */
              xsend((char*)p,node,0);   /* If count != 0, pass it on.    */
          else xsend((char*)p,HOST,0);  /* If count == 0, back to host. */
    }
}
```

This program can be run for large numbers of hops and for various message lengths to characterize the average message latency of different systems under conditions in which the network is uncongested and all nodes are equally likely message destinations. The time measured by the program includes a round trip between the host and node 0, which can be measured by running the host program with a "hops" argument of 0. This latency may then be subtracted from other measurements. When this program is run on the following systems, it yields these measurements of the average time to send an L byte message:

`ghost` On a Sun3 ghost cube, independent of the ghost cube size, the average time per message is $\approx(20,000+16L)\mu$s with nodes distributed across a network; $\approx(20,000+2.5L)\mu$s with all nodes on one Sun3/160; and $\approx(14,000+1.5L)\mu$s with all nodes on a Sun3/260.

The constant term in these expressions is the fixed overhead of the `xsend` and `xrecvb` functions, often referred to as the *software overhead*, while the coefficient of L reflects the bandwidth of the communication.

`cosmic` On a Cosmic Cube running RK, the average time per message can be expressed as $\approx(5000+40L+(500+7L)D)\mu$s, where D is the average message distance in a binary n-cube.

`ipsc` On Intel iPSC/1 systems running the NX operating system, but whose node processes have been compiled with the `cube` library functions described in this guide, experiments with various size cubes and messages show that the average time per message can be expressed as $\approx(1900 + 1.0L + (500 + 0.2L)D)\mu$s.

The fixed part of the dependence on the message distance on these very similar binary "hypercube" machines is due to software that examines the header and controls the route in each node, and the length-dependent part is indicative of a residual component of store-and-forward routing of flow-control units or packets. The largest performance difference between the Cosmic Cube and iPSC is seen in the coefficients of L; this is because the Cosmic Cube wormhole routing routines process an interrupt for each 8-byte flow-control unit, whereas the Intel iPSC/1 cut-through routing routines use DMA channels to transport packets.

The following measurements were made for this April 1989 revision of the "C-guide" on 16-node Intel iPSC/2 and 64-node Symult Series 2010 multicomputers operated by our project group.

`ipsc/2` For the Intel iPSC/2 using the NX 2.4 node operating system, but whose node processes have been compiled using the `cube` library functions described in this guide, the average time per message is $\approx(675 + 0.2L)\mu$s when $L \leq 100$, and $\approx(1320 + 0.48L)\mu$s when $L > 100$. When the same program is expressed in terms of the NX 2.4 primitives, the average time per message is $\approx(413 + 0.2L)\mu$s when $L \leq 100$, and $\approx(751 + 0.36L)\mu$s when $L > 100$. The iPSC/2 employs a different protocol for messages that convey a payload of 100 or fewer bytes. The extra time required to simulate the CE/RK primitives in terms of the NX 2.4 primitives is due to an additional function call, `malloc` and `free` operations, and one extra copy operation.

`s2010` The corresponding expression for the message latency of the Symult Series 2010 using RK as the node operating system is $\approx(177+0.11L)\mu$s. The detail structure of the latency as a function of L reveals that the Series 2010 fragments messages that are longer than 256 bytes into a sequence of 256-byte packets. This fragmentation and reassembly is done in order to prevent long messages from blocking short messages for long periods of time. The latency for $N \leq 256$ is $\approx(177 + 0.05L)\mu$s.

Because these second-generation systems use hardware wormhole routing, there is no component of store-and-forward routing. In fact, the distance-dependent term is so small that it could not be measured reliably using the example program. The largest part of the message latency is the software overhead. Substantially smaller software overhead and latency are achieved by systems and programs that run at the handler level (see Appendix D).

10. Examples with Message Types

It often happens in writing multicomputer application programs that a process must distinguish between a number of different types of messages. A particular field of all messages can be reserved to carry a *message type*. Even though the number of types is typically small, this field is usually reserved as the first `long`, so as to retain word alignment of the structures that follow.

One common programming paradigm in using message types is to dispatch according to the type immediately on receipt of each message:

```
#include <cube/cubedef.h>

unsigned long *p;

main()
{
    while(1) {
        switch(*(p = (unsigned long *) xrecvb())) {
            case  0: method_0(p+1); break;
            case  1: method_1(p+1); break;
            .
            .
            default: print("No such method."); break;
        }
        xfree((char *)p);
    }
}
```

This framework for a process is reminiscent of "method lookup" in object-oriented programming. Each message received can be thought of as causing a specific action (invoking an attribute or method) in the process (object). A pointer to the rest of the message is passed to the function that is selected in the dispatch. The reactive primitives are clearly tailor-made for such situations.

In other situations it may be more convenient for a process to deal with messages of different types in a specific order. Here it is convenient to define a different set of message functions that allow the process to *exercise discretion* in the order in which messages are received. These message functions work by maintaining separate input queues for each message type.

The convention used in the macros and functions that follow is that the first `long` of a message holds the type field. Referring to the typed message only via a pointer to the location following the type field hides the type information. For each of the "x" functions, a similar "t" function is defined. This technique of *layering* customized message functions on top of the "x" primitives can be carried to much greater length than in this example. The Fortran primitives described in Appendix E are a set of library routines similar to those shown here, but which include both the type and the sender ID in the message, and which allow the programmer to

filter messages directly into Fortran arrays by type and/or sender ID.

The "t" functions are so similar to their corresponding "x" functions that several of the "functions" can be defined as macros. A `ttype` function, analogous to `tlength`, is included to return the hidden message type.

```
#define tmalloc(n) (((long *) xmalloc(n+sizeof(long)))+1)
#define tfree(m)   (xfree(((long *)m)-1))
#define tlength(m) (xlength(((long *)m)-1)-sizeof(long))
#define ttype(m)   (((long *)m)[-1])
```

The `tsend` function is the same as `xsend` with the addition of a message type as the last argument. In order to ensure that this function can also be used in the host environment, the `htocl` byte-order conversion function is used on the type field.

```
tsend(m,n,p,t)                          /* Send a message buffer to   */
    long *m;                            /* process (n,p) with type t. */
    int n,p,t;
{
    *--m = t;                           /* Put the type into first    */
    htocl(m,1);                         /* long word, byte order      */
    xsend(m,n,p);                       /* convert it, then send it.  */
}
```

The `trecvb` function calls functions `pull_q(type)` and `push_q(type)` (not shown) that do the queue management. Of course, these routines do not store the actual messages, but only the pointers to them. If the maximal number of queued messages is small, these routines can reasonably be written to keep the pointers in a simple list that is searched by the `pull_q` function and appended to by the `push_q` function.

```
long *trecvb(t)                         /* blocking receive will not  */
    int t;                              /* return until a message of  */
{                                       /* type t is received.        */
    long *m;

    if(m = pull_q(t)) return(m+1);      /* Check private queue for msg*/
                                        /* of type t.                 */
    while(m = (long*) xrecvb())         /* Repeatedly wait for a msg  */
    {                                   /* to arrive, convert byte    */
        ctohl(m,1);                     /* order of the type field    */
        if(*m == t) return(m+1);        /* and return only if the     */
               else push_q( m );        /* type of the message        */
    }                                   /* is equal to t.  Else,      */
}                                       /* save the message.          */
```

The `trecv` function is quite similar to `trecvb`. It is interesting that the `while` in the `trecv` function could be replaced correctly by an `if`. The function is written in such a way that it will not return NULL unless its internal `xrecv` call returns NULL, even if it has to

move all of the messages queued for this process out of the system's receive queue and into its own private queue. If, however, the process were polling for several message types, it might be more efficient to replace the `while` by an `if`, so that the function will return after moving no more than one message from the system's queue to its private queue. The weak semantic specification of `trecv` (`xrecv`) permits NULLs to be returned in such cases (see the digression in Appendix A).

```
long *trecv(t)                          /* Nonblock receive returns 0 */
    int t;                              /* if no msg of type t.  It   */
{                                       /* returns a msg of type t    */
    long *m;                            /* if one is found.           */

    if(m = pull_q(t)) return(m+1);      /* See if there is one already*/
                                        /* in private message queue.  */
    while(m = (long*) xrecv())          /* Otherwise check system's   */
    {                                   /* input msg queue.  If yes,  */
        ctohl(m,1);                     /* Byte order convert the type*/
        if(*m == t) return(m+1);        /* field and return the msg   */
               else push_q( m );        /* if it is type t, otherwise */
    }                                   /* save it in private queue.  */

    return(0);                          /* Return 0 if not found.     */
}
```

11. Concurrent Formulations

While this guide is meant to focus on the mechanics of writing and running programs in the CE/RK programming environment, let us conclude with a description of some of the considerations for formulating concurrent programs for multicomputers.

Although the functions provided in this low-level portable environment can be used as a compilation target for higher-level concurrent programming notations, the low-level environment built on the C programming language is adequate in itself for many purposes. The CE/RK programming system described herein has been used extensively in experiments with concurrent algorithms, and in writing useful programs that are based on explicit concurrent formulations of computing problems typical of those found in science and engineering. These application programs are based, for example, on concurrent adaptations of well-known sequential algorithms, on regular data-partitioning strategies, on the systolic algorithms that have been developed for regular VLSI computational arrays, and on a wide variety of fundamental concurrent algorithms.

The multiple-process message-passing model of computation and the primitives of the CE/RK environment are based on a principle of a "separation of concerns" between (1) the expression, in a collection of processes, of an explicit concurrent formulation of a computing problem; and (2) the distribution, or *placement,* of processes into the nodes and hosts. The system thus encourages "late binding" of processes to nodes, so that this decision can be deferred until the interdependence and computational demands of different processes are known. The decision can even be deferred to runtime choices made at the moment that a process is to be spawned.

The performance of a particular program depends on how well the concurrent formulation and process placement keep the nodes busy. The usual objectives are to assure that: (1) The number of concurrent processes that are able to make progress is on average comparable to or larger than the number of nodes, and (2) there is a way to distribute the processes so that the average computational load on the nodes is satisfactory. It is possible (although difficult) to formulate a concurrent computation that satisfies the first but not the second requirement. One should accordingly consider the "load-balancing" properties of the computation from the earliest stages of program design.

In addition to this primary consideration, which is fundamental to the architecture, a second performance consideration is that messages should be reasonably infrequent compared with computation, as there is an appreciable operating system overhead for each message.

The programmer should not be overly concerned about the placement of processes to minimize message distance or message contention in the message network. This consideration is so unimportant in second-generation multicomputers that the programmer does not even need to know what the network topology might be. Applications that produce extremely high volumes of messages will benefit somewhat from process placements that localize message traffic on very large configurations of second-generation multicomputers; however, analytical results, statistical studies, and simulations show that even after the constraints noted above have been met, it is always possible to find a process placement that will reasonably minimize both the distance a message must travel and the contention in the network.

Appendix A: Implementation Idiosyncrasies and Other Digressions

Much of what is in this Appendix could have appeared as footnotes, but, since we *hate* footnotes when we are reading, we restrained ourselves to the extent of placing the footnotes here:

Section 2: Generalization of the `getcube` *Utility*

We plan to augment the `getcube` utility program to allow a more general specification of the number and types of nodes in heterogeneous multicomputer systems.

Section 3: Restrictions on Process Placement

Current multicomputer nodes are not virtual memory machines; hence, process placement is constrained by the storage requirements of processes and by the total storage available in each node. Ghost cubes support virtual memory in the nodes, and the second-generation multicomputers are expected in the future to support virtual memory in the nodes.

Section 4: Digression on Scheduling Properties

If the description of the scheduling properties of the `xrecv` function strikes the reader as impossibly vague, let us point out that the specification is weak *deliberately* so that different implementations of RK may employ different scheduling strategies, yet still satisfy the same weak semantic specification. For example, the `xrecv` function is permitted to return a NULL pointer at any time. In current implementations, it will not do so unless there are no messages queued for any process in the node. If there are no queued messages in a node, the kernel will return NULL pointers in a round-robin fashion to all processes that have called `xrecv`.

Similarly, an implementation of RK that has been tuned for a node with a relatively large context-switching overhead might not force a context switch on an `xrecv` or `xrecvb` if the next message is for a different process, but instead might first conduct a limited search into the receive queue for the next message for this process. If such a message is found, the context switch is avoided. Although the scheduling is described as message driven, the search down the receive queue falls within the specifications for the operation of the message system because message order is specified as being maintained only between *pairs* of communicating processes.

The purely reactive form of scheduling is bypassed also in dealing with exceptional events, for example, priority messages for system services, such as process spawning, and system errors. It is also bypassed to provide a remote procedure call (RPC) mechanism.

The primitive message functions `xsendrpc` and `xrecvrpc` have the same usage as `xsend` and `xrecv`, but an `xrecvrpc` function will not return until a message sent by an `xsendrpc` function arrives. A remote procedure call can be implemented by the caller sending a message to a process that acts as a remote procedure. When the return value is required, the caller executes an `xrecvrpc`. This function does not return until the reply message arrives, *independent of the arrival of other messages.* The process that acts as the remote procedure accepts the message as usual with `xrecvb`, and replies with `xsendrpc`. These functions, and the messages tagged as RPC messages, are used by functions such as `print` and `execute`, and by the standard I/O package, and would wisely be avoided by the casual user. The appearance of a function such as `print` between an `xsend/xrecvrpc` pair may lead to the reply messages being interchanged, leading to very curious bugs.

The standard I/O package is a pretty demonstration of the use of the `xsendrpc` and `xrecvrpc` message functions. Interested users may wish to study the source code of the standard I/O package for examples of the use of these functions.

Section 4: Digression on Broadcast Messages

Certain systematic patterns of broadcast and combining of messages are common in user programs, and can be accomplished very efficiently in the RK at the "handler" level (see Appendix D). We have defined a set of functions for the "extended RK" (which is currently being written) that provide global

operations across sets of cohort processes. These global operations include barrier synchronization, sum, min, max, prefix sum, and rank.

A closely related issue is an appeal from users for a "non-destructive" send operation. `xsendb` and `xmsendb` are experimental functions in the CE library. They do not return until the message is sent (or at least so it would appear to the user), and the message buffer is not freed. In fact, the message buffer referred to in `xsendb` and `xmsendb` is not restricted to being a dynamically allocated message block. These functions have an additional `length` (in bytes) parameter:

```
xsendb(p, node, pid, length);

xmsendb(p, cnt, destinations, length);
```

Any sequence of bytes, whether an array, a part of an array, or a part of dynamically allocated memory, can be sent as a message.

Section 5: Other Functions

With regard to the standard I/O package (`stdio`), all operations on the structure `FILE` are allowed. Notice the distinction between the class of functions that operates on `FILE`s, such as `fopen`, and the other class that operates on *file descriptors*, such as `open`. The first is implemented on top of the second; however, since not all of the second are needed, not all members in the second class are implemented. The ones that are implemented are used in conjuction with the secondary server process, `FILE MGR`, which is started automatically during `getcube` on multicomputers on which this package is supported.

Caveat: It is possible in this implementation for one node process to pack up a file handle and send it in a message to another process, but we are not sure (due to distributed file caches) that this will be legal for systems that have their own built-in file systems.

Section 6: Compiler for Cosmic Cube and iPSC/1 Node Processes

The lack of memory protection in the 8086 processors in the Cosmic Cube nodes and the limitations of the compiler used for both the Cosmic Cube and iPSC/1 make it necessary to pay attention to a process's stack, which is used for automatic variables. If the `-F` option to `cccos` or `ccipsc` is omitted, the stack will default to a maximum size of 2K bytes. If the process requires a larger stack than the 2K-byte default, the `-F` option should be used. Its argument is a *hexadecimal* number that specifies the maximum number of bytes of stack that will be required by the program at any time.

Appendix B: How To Make a Ghost Cube

The *ghost cube* feature of CE allows one to treat a group of identical and NFS-connected 4bsd UNIX machines as a multicomputer. Ghost cubes accurately emulate the CE/RK environment as it runs on multicomputers.

The "cube" is connected to the CE system via the `ifc` process `ghost_ifc`. The `ghost_ifc` boots the cube by reading a list of usable hosts from a file and starting a `ghost_rem` process for each simulated node, using each host in turn. A typical invocation of `ghost_ifc` may look like:

```
% ghost_ifc -d 3 -s host_list &
```

Running `ghost_ifc` in this way starts a 2^3-node ghost cube that is *private* to the user, and the name of the ghost cube is the user's UNIX username. (A ghost cube in which N is not a power of two can be started, *eg,* by `ghost_ifc -d 3n ...` for 3 nodes.) The file `host_list` contains a number of *left-justified* entries of this form:

```
neptune    /usr/cube/ghost_rem
mercury    /usr/cube/ghost_rem
saturn     /usr/cube/ghost_rem
mars       /usr/cube/ghost_rem
 wraps around from here
```

Lines starting with a blank are treated as comments. The first entry in each line is the name of the host; the second entry specifies the program to run on the remote machine. There may be more or fewer entries than are needed to fill the specified cube size. The `ghost_ifc` will "boot" the ghost cube by going down the list sequentially, skipping those hosts that fail `rsh`, and wrapping around if it hits the end.

After the ghost cube has been booted, it will appear as an entry in the cube dæmon's table (see page 5), just like all other cubes. All cube-independent utilities and host programs can be run from anywhere on the network. However, it is advisable to do the `getcube` in the same NFS domain as the `ghost_ifc` because the first thing the `ghost_rem` process does after a `getcube` is to `cd` to the working directory of the `getcube` process. Process spawning in a ghost cube does not involve the transfer of any programs as messages, as it does in real multicomputers; instead, the name of the program is given to `ghost_rem`, which runs the program out of its working directory.

One can also make a *public-access* ghost cube by giving it a name with the `-n` option. If the ghost cube is to reboot itself if killed, it can be run by preceeding the command with the `loop` program:

```
% loop ghost_ifc -n "mimic d3" -d 3 -s host_list
```

The `loop` program goes into the background, and restarts the `ifc` process if it is killed. The name of a public ghost cube starts with a colon (:), and all users on the network can allocate it. When a `ghost_ifc` is kept running all the time like a normal dæmon, it should for reasons of file protection be run by a special user whose only purpose is to run the `ghost_ifc`. Note: Because the `ghost_rem` processes access the user's files directly, the user's files must not be protected from the special user that runs the `ghost_ifc` for a public-access ghost cube. Similarly, when using the `stdio` package on ghost cubes, it is necessary for users to make their files readable and/or writable to this special user.

Appendix C: Host Utilities

The first class of host utility programs includes those that interact with the CE, and whose usage is identical for all multicomputers. These utilities include the `peek`, `getcube`, and `freecube` programs described in sections 2 and 7. Another such utility is `killcube`, which may be used in place of `freecube` to kill the `ifc` process if there is no other user of the allocated multicomputer. The `ifc` process is normally run with the `loop` program (Appendix B), which will reboot the multicomputer if the `ifc` process is killed. There is also a `restart` utility that is used to kill the cube dæmon (`cubed`), but the use of this utility in most installations is restricted to system managers.

Another class of utility programs includes those that, if implemented at all, have different implementations for different multicomputers, but generally have the same usage. These utilities include:

Function	non-cube	ghost	Cosmic	*iPSC/1*	*iPSC/2*	*S2010*
Process spawn	`spawn`	`spawn`	`spawn`	`spawn`	`spawn`	`spawn`
Process kill	`ckill`	`ckill`	`ckill`	`ckill`	`ckill`	`ckill`
Process status	—	`cps`	`cps`*	—	—	`cps`
Load average	—	—	`cla`*	—	—	`cla`*
Remote LEDs	—	`ledpanel`	—	`ledpanel`	`ledpanel`*	`ledpanel`*
C compiler	`cch`	`ccgh`	`cccos`	`ccipsc`	`ccipsc2`	`ccs2010`
Archiver	`arh`	`argh`	`arcos`	`aripsc`	`aripsc2`	`ars2010`
Lint	`linth`	`linth`	`linth`	`linth`	`linth`	`linth`
Namelist	`nm`	`nm`	—	`nmipsc`	`nmipsc2`	`nm`
Debugger	`dbx`	`cdbug`	—	—	—	`mdbx`

* Implementations may be absent or experimental.

The `spawn` and `ckill` utilities were described in section 3. The usage of `cps` is described by running `cps -u`. The display produced by `cla` is self-explanatory.

Remote LED display: For those persons who have become addicted to watching the LED displays that decorate most multicomputers, a remote display can be obtained on a Sun workstation by running:

```
% ledpanel sunled
```

from within a Sun window after running `getcube`, but before spawning any node processes. The LED display that is generated on a separate window contains three LEDs per node, and the window can be resized to personal preference. Maintenance of this display requires quite a lot of communication if the multicomputer is large, and the display may not reliably track the real LED display.

Archiver: These utilities can be used to build C libraries. Although some of them will accept the full set of UNIX archiver options, the only option string that is universally supported is the `cuv` string.

Lint: Since the header files for CE programs usually are not stored in the normal UNIX header file directory (`/usr/include`), compilers for CE programs contain specific instructions to reference the CE header files in addition to those contained in `/usr/include`. The `linth` utility is the same as the UNIX `lint` except that it also looks for the CE header files.

Namelist: The structure of an object code file for a node program is often different from object code files for the host on which the node program is maintained. In order to display the symbol

table of these programs, a nm utility is provided for each type of object code that the environment supports. The only defined usage for these nm utilities is that each takes the name of one node program as the argument, such as:

```
% nmipsc ipsc_program
```

Debugger: The cdbug utility is available in some machines for debugging node programs. This utility should be run before the node program to be debugged has been spawned.

% cdbug *node pid [optional alternate debugger program]*

The cdbug program should be run from a window of its own on a window system because it ties up its tty input reading debugger commands. On a Berkeley UNIX system in which job control is allowed, you can start the cdbug initially in the background and bring it to the foreground when debugging is in progress. You can debug multiple-node programs by running one cdbug program for each node program. On a non-window system, however, debugging multiple-node programs can get out of hand very quickly.

Once invoked, the cdbug program will not start taking commands until the spawn command for the specified process has been issued. On a ghost cube, the corresponding spawn operation will not complete until the process is actually created by the debugger and executed up to the main procedure. Following is an example of how this might be done on a Berkeley UNIX machine using dbx:

```
(dbx) stop in main
[1] stop in main
(dbx) run
[1] stopped in main at ....
(dbx)
```

On some hosts, cdbug will automatically insert these two commands so that the program will have stopped in main before cdbug starts taking commands from the terminal.

Appendix D: Handlers

RK and CE are structured internally to be able to support any of a variety of message primitives. This structuring conforms to a particular discipline of reactive handling of messages.

The processing associated with the arrival of a message usually consists of queueing the received message in memory for the process to which it is addressed. Alternative actions may also be invoked, including discarding the message, sending an acknowledgment message, or using the message to create a code or data segment for a process being spawned. Each message delivered must be sufficiently processed that future messages are not permanently blocked from delivery. The property of freedom from deadlock for the routing network is dependent upon this minimal processing; every message that is delivered to a node must be consumed.

Each message is given a *tag* as it is sent, and this tag determines the method for handling the message when it is received. The tag is used as an index into a table in the inner kernel to select a kernel process called a *handler*. The table entry contains an entry point for that handler's code and a pointer to a data segment for the variables that persist between invocations of the same handler. Handlers are able to modify the handler table to change their entry point, are able to send messages, and are able to create new handlers. The crucial property of a handler is that it will not execute indefinitely.

Within this fundamental framework, one may build a variety of handlers. Some of the handlers in RK are devoted to system functions, such as initialization and the spawning of new handlers. Each of the handlers for user messages and system calls is associated with one or more libraries of functions that provide the application programmer with a set of process-spawning, message, and other primitives. Given only a handler and library, the CE/RK environment running on a "brand X" multicomputer is able to run programs written in terms of the primitives of a "brand Y" multicomputer.

It is generally not the job of the application programmer, but of the system programmer, to write handlers. Rather, the application programmer may choose a "standard" handler whose message functions are convenient for the application at hand. Programs are compiled using a library that is specific to this handler, and a specified handler is loaded into the set of multicomputer nodes when those nodes are allocated to a user.

There has so far been only one "standard" handler written for the CE/RK environment. The message-handling actions, system calls, and scheduling supported by this *reactive* handler are analogous to the internal interface between the inner sections of RK and handlers. A technical report "The Reactive Kernel" (Caltech-CS-TR-88-10) describes the internal structure of RK, and this report should be consulted by adventurous users interested in implementing their own handlers.

There are currently two "standard" libraries associated with the reactive handler. The library for the reactive message primitives and other functions described in this guide is, for historical reasons, called `cube`. There is also a compatibility library for the functions that have been in use for the past five years on the Cosmic Cubes.

Appendix E: Proposed Standard Fortran Interface

The following "standard" CE/RK Fortran interface routines were defined based on extensive discussions between Caltech and Symult programmers, and were implemented by Symult Systems on their Series 2010 multicomputer. Symult Systems has kindly given us permission to distribute the portable library version of these Fortran primitives with CE under the usual non-redistribution agreement (see page 34).

The core of the Fortran interface routines are sends, blocking receives, non-blocking receives, and the spawn and kill routines. Subroutines in this set are nominally prefixed with `f`. Integer functions are prefixed with a letter in the range `i`–`n`.

The header file `fcdef.h` defines the types of the non-UNIX functions, and the values of parameters `ALL`, `ANY`, `FALSE`, `HOST`, and `TRUE`. This header file should be included in any source code whose compiler allows the inclusion of header files; otherwise, users should copy the contents of this header file into each source module.

With respect to the subroutine and function parameters, `len`, `type`, `node`, and `pid` are both input and output parameters; therefore, they must be variables, not constants. `buf` may specify any variable or array that may be equivalenced to a variable of type integer. `cbuf` may specify any character variable or array. The `type` variable may be an integer value in the range 0 through 65535. The `node` variable may contain a value specifying any node number. Nodes are numbered 0 through N-1, where N is the number of nodes assigned to the process group. Except as noted, the `node` variable may also contain the value `HOST`, to indicate a host process. The `pid` variable may contain any value accepted by the C version of the Reactive Handler.

There are two classes of send and receive calls: those for data objects that occupy numeric storage units, and those for data objects that occupy character storage units. Care should be taken by the application programmer that data sent by one class of call is not received by the other class. Types should be used judiciously to ensure this separation. In particular, when a program is using both classes of calls, the same type that is assigned to a numeric message should not be assigned to a character message, and the receive call should avoid using the parameter `ANY` for the receive type.

Subroutines

```
subroutine fsend(buf, msglen, type, node, pid)
integer buf, msglen, type, node, pid

subroutine fsendc(cbuf, msglen, type, node, pid)
character*(*) cbuf
integer msglen, type, node, pid
```

The `fsend` and `fsendc` subroutines are respectively numeric and character sends that transmit the first `msglen` bytes in `buf` or `cbuf` as a message of the specified `type` to `(node,pid)`.

```
subroutine frecv(buf, len, type, node, pid)
integer buf, len, type, node, pid

subroutine frecvc(cbuf, len, type, node, pid)
character*(*) cbuf
integer len, type, node, pid
```

The `frecv` and `frecvc` subroutines are respectively numeric and character blocking receives that exercise discretion by `type`, `node`, and `pid`, some or all of which may be specified as `ANY` to indicate no preference. `buf` or `cbuf` is output; `len`, `type`, `node`, and `pid` are input/output. Therefore, when the subroutine is called, these input/output variables have values the specify the characteristics of the message to be received; after the subroutine returns, these input/output variables have values

corresponding to the message received. For example, frecv might be called with the values of node and pid being ANY; on the return, the values would be those corresponding to the ID of the process that sent the message. The input value of len is the number of bytes available in buf or cbuf to receive the message. No more than len bytes are transferred to buf or cbuf, even if the message must be truncated, but the output value of len reflects the actual length of the received message.

```
subroutine fkill(node, pid)
integer node, pid
```

The fkill subroutine terminates the specified node process. The node parameter may be ALL to indicate that all node processes with the specified pid are to be killed; node may not be HOST. Similarly, the pid parameter may be ALL to indicate that all node processes in the specified node are to be killed. Both node and pid may be ALL to terminate all node processes.

```
subroutine fled(value, i)
integer value, i
```

When used by a node process, the fled call turns the ith LED (or other output device) of this node on or off, based on the input parameter value. The LED is turned on if value is TRUE and off otherwise. The functioning of this call when used by a host process is left to the discretion of the system implementers.

```
subroutine finit(node, pid)
integer node, pid

subroutine falias(alias)
character*(*) alias
```

When used by a process running on a host, the finit call assigns the specified (node, pid) ID to this process, and the falias call designates a character string, alias, to be used in place of the name of the calling process when the status of the process group is inquired about (by use of the peek or cps programs). The default name for a process is the name by which the process was invoked, typically its file name. If the finit call is used with the value of the node parameter other than HOST, the user is responsible for ensuring that no process with the same (node,pid) is running on a node. If present, the finit call must precede the use of all other CE system and library calls, except falias. Both calls are null operations when used by node processes.

```
fprint(string)
character*(*) string
```

The fprint subroutine causes the formatted message in string to be sent from this process to the (HOST,SERVER) process, which puts the message on the user's screen.

```
subroutine exit(status)
integer status
```

A call to exit terminates the process; control does not return from this call. When used by a host process, the functioning of this call is identical to that of the UNIX call of the same name; status should be in the range 0–255. The meaning of the status parameter when this call is used by a node process is left to the discretion of the system implementers.

Functions

```
integer function ispawn(file, node, pid)
character*(*) file
integer node, pid
```

The ispawn function loads an executable image specified by the file parameter into the specified node, creates a process for that image, assigns to it the specified pid, and causes the process to begin execution. file is a character argument that specifies a file relative to the working directory of the getcube program that allocated the process group. node may be ALL to indicate that the process should be spawned into every node; node may not be HOST. The value of the function is zero for success, and a non-zero error code for failure.

```
integer function mynode()
integer function mypid()
integer function nnodes()
integer function ndim()
```

The mynode, mypid, and nnodes functions are the same as the C functions of the same name, and ndim is the same as the C function cubedim.

```
integer function irecv(buf, len, type, node, pid)
integer buf, len, type, node, pid

integer function irecvc(cbuf, len, type, node, pid)
character*(*) cbuf
integer len, type, node, pid
```

The irecv and irecvc functions are respectively numeric and character *non-blocking* receives that exercise discretion by type, node, and pid, some or all of which may be specified as ANY to indicate no preference. buf or cbuf is output; len, type, node, and pid are input/output. If the value of the function is non-zero, then the desired message was not available, and the parameters are unaltered from their input values.

```
integer function iexcut(string)
character*(*) string
```

The iexcut function causes the formatted message in the variable string to be sent from this process to the (HOST,SERVER) process, which uses a *UNIX* system call to perform the command specified in the formatted message. The value of the function is the value returned when (HOST,SERVER) does the system call.

```
integer  function iclock()
```

The iclock function returns the value of a counter, modulo 2^{31}, which is incremented at approximately 1ms intervals.

UNIX Routines

Typical implementations will also include a UNIX Annex, together with the usual routines found in UNIX Fortran libraries.

Appendix F: Historical and Bibliographical Notes

The design of the programming system described in this guide was based in largest part on an exchange of ideas between Wen-King Su, William C. Athas, and Charles L. Seitz during 1986. The first implementation of the Reactive Kernel was written in 1987 by Jakov Seizovic, and the version 7 Cosmic Environment was written in 1987 by Wen-King Su.

The following papers and reports from our research group, listed in reverse chronological order, contain additional information about multicomputer architecture and programming. Caltech Computer Science technical reports may be ordered from the Computer Science Librarian, M/S 256-80, California Institute of Technology, Pasadena CA 91125, at the prices listed.

Wen-King Su and Charles L. Seitz, "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm," Caltech-CS-TR-88-22 ($2.00); also published in the *Proceedings of the 1989 Eastern Multiconference, Distributed Simulation Conference,* ACM/IEEE, 1989. *Distributed discrete-event simulation techniques and performance results using CE/RK multicomputers.*

Nanette Boden, "A Study of Fine-Grain Programming Using Cantor," Caltech-CS-TR-88-11 ($5.00). *This MS thesis contains numerous examples of Cantor application programs together with a critique of Cantor 2.0.*

Jakov Seizovic, "The Reactive Kernel," Caltech-CS-TR-88-10 ($3.00). *This MS thesis explains the internals of the Reactive Kernel.*

William C. Athas and Charles L. Seitz, "Multicomputers: Message-Passing Concurrent Computers," *IEEE COMPUTER* 21(8), pp 9–24, August 1988. *A status report on multicomputers as of mid-1988.*

Charles L. Seitz, William C. Athas, Charles M. Flaig, Alain J. Martin, Jakov Seizovic, Craig S. Steele, Wen-King Su, "The Architecture and Programming of the Ametek Series 2010 Multicomputer," *Proceedings of the 1988 Hypercube Conference,* ACM, 1988. *A concise description of the design of the Symult Series 2010 Multicomputer.*

William C. Athas, "Fine Grain Concurrent Computation," Caltech-CS-TR-87-5242 ($8.00). *This PhD thesis contains the definition and semantics of Cantor 2.0, compilation techniques, and numerous examples.*

William J. Dally, *A VLSI Architecture for Concurrent Data Structures,* Kluwer Academic Publishers, 1987. *This PhD thesis contains the definition of Concurrent Smalltalk, examples of message-passing programs and algorithms for concurrent data structures and graph problems, and an analysis of message-passing network performance showing that low-dimension networks are optimal for medium-grain multicomputers.*

Charles L. Seitz, "The Cosmic Cube," *Communications of the ACM,* vol 28, no 1, pp 22–33, January 1985. *The standard reference on the Cosmic Cube experiment.*

Charles L. Seitz, "Concurrent VLSI Architectures," *IEEE Transactions on Computers,* vol C-33, no 12, pp 1247–1265, invited paper for the *Centenial Issue,* December 1984. *A tutorial article on the message-passing architectures inspired by VLSI technology, including the Connection Machine, systolic arrays, and multicomputers.*

Charles R. Lang, "The Extension of Object-Oriented Languages to a Homogeneous, Concurrent Architecture," Caltech-CS-TR-82-5014 ($15.00). *This PhD thesis contains much of the basis for the design and programming method of the Cosmic Cube experiment.*

Cosmic Environment Software Distribution

The Cosmic Environment software described in this guide is available for internal use to most research organizations under the arrangements outlined below. Other versions of this software are available through Intel Scientific Computers and Symult Systems, Inc.

This software also comes with a network-based update mechanism that can be used to distribute it conveniently between hosts on TCP/IP networks. This mechanism is the recommended way of obtaining a distribution for Internet hosts.

To get a copy, send a letter to the undersigned that includes:

1. a statement of your intended use of this software (brief);
2. your agreement that you will not redistribute this software outside of your organization, and will keep the source files protected on your systems;
3. your agreement that this software is for your internal use, and that you will not sell access to or otherwise permit use of these programs to people outside your own organization; and
4. your understanding that this is experimental, unsupported software that may or may not work as described, and that you hold Caltech harmless from any loss or damage that may result from its installation or use.

The letter must be from a person responsible for the research group; for example, in a university research group, the letter should be from a faculty member rather than a graduate student.

We are able to answer questions about this software as our time allows, but only by electronic mail to `cube@vlsi.caltech.edu`. Bug reports are also welcome. It is most helpful if a bug report includes the smallest example of code that you can devise that demonstrates the bug.

If you are installing this software on a machine that is not an Internet host, or prefer to use tape, please include with your letter a blank 600' 1/2-inch tape (no backcoated tape, please) labeled with your postal address. We will mail the C sources to you in UNIX `tar` format at 1600 bpi (default) or 6250 bpi (by request). The files can be placed in any source directory. The installation instructions are included in a `README` file.

If you are installing this software on an Internet host, please do not send a tape. Just send us your Internet mail address and the name of the host on which you intend to maintain your source directory. We will then enter the name of the host on which you maintain the sources into a permission table, and will send you a message that includes a C program that you can compile and run to bring the sources over the Internet to your machine. This program makes a socket connection to a Caltech host, and brings the files across the network. This same program can be used later for automatic updates, which are selective by timestamp in order to minimize the volume of messages sent through the network. *(If you are uncertain whether the machine you specify is an Internet host,* try to telnet to `csvax.caltech.edu`. If you are successful, then all is well.)

Charles L. Seitz
Computer Science 256-80
California Institute of Technology
Pasadena, CA 91125